3. NPM & Create React App

Single Page Application (SPA)

Modern web applications are also called Single Page Applications, so what does it mean?

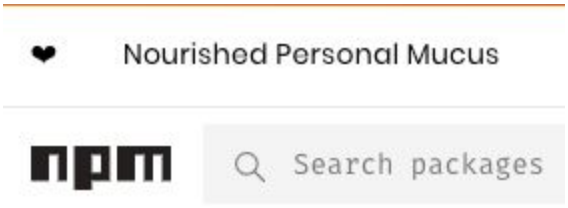
In the past, when browsers were much less capable than today and JS performance was poor, every page was coming from a server. Every time you clicked on some element, there was a new request created and the browser subsequently rendered the new page.
Single page application is a principle, where you load application code (HTML, CSS, JS) only once. When you interact with a browser, instead of a request to the server that returns a new document, the client (browser) requests some JSON information or performs an action on the server, but the page that you see never really wipes away. This principle (popularized by modern front-end frameworks like React, Angular…) is built in JavaScript or at least compiled to JavaScript and works in the browser, but behaves more like a desktop application.

Example of SPA

- Gmail
- Google Maps
- Google Drive
- Google Docs
- Facebook
- Twitter

A Single Page Application feels much faster to the user, because instead of waiting for a server to rerender your page, you can have instant feedback in UI. This solution is ideal for building a complex UI with many control components, like buttons, inputs and so on.
The main con of SPA is how heavily it relies on JavaScript, which means that it can run slowly on some older devices. Also the user can be so cautious that they disable the running JS in their browser and that is when SPA stops working.

NPM and NPX



In lesson 2 we successfully installed Node.js environment, hopefully... Now we can move forward to Nourished Personal Mucus... No, just kidding, it's **Node Package Manager**.

Npm is the world's largest software registry, similar to the composer in the PHP world. Open source developers from all around the world use npm to share and use packages, and many organizations use npm to manage their private development as well.
The package.json file is a key element that serves as a manifest of an application. It contains various metadata that are relevant to the project.
For more information please visit https://docs.npmjs.com/.

From version npm@5.2.0 there is alongside NPM also installed a binary called NPX. NPX is a tool intended to help round out the experience of using packages from the npm registry — the same way npm makes it super easy to install and manage dependencies hosted on the registry, npx makes it easy to use CLI tools and other executables hosted on the registry. We will use NPX to execute our first command (CRA), to create the basic structure of our SPA.

Create React App

Move to your projects folder and run the following command in CLI.

```
npx create-react-app my-app
```

After a successful installation, `my-app` folder has predefined basic structure of React SPA. We will look closer at the main parts.

```
my-app
```

```
├──── README.md
├──── node_modules
├──── package.json
├──── .gitignore
├──── public
│     ├──── favicon.ico
│     ├──── index.html
│     └──── manifest.json
└──── src
      ├──── App.css
      ├──── App.js
      ├──── App.test.js
      ├──── index.css
      ├──── index.js
      ├──── logo.svg
      └──── serviceWorker.js
```

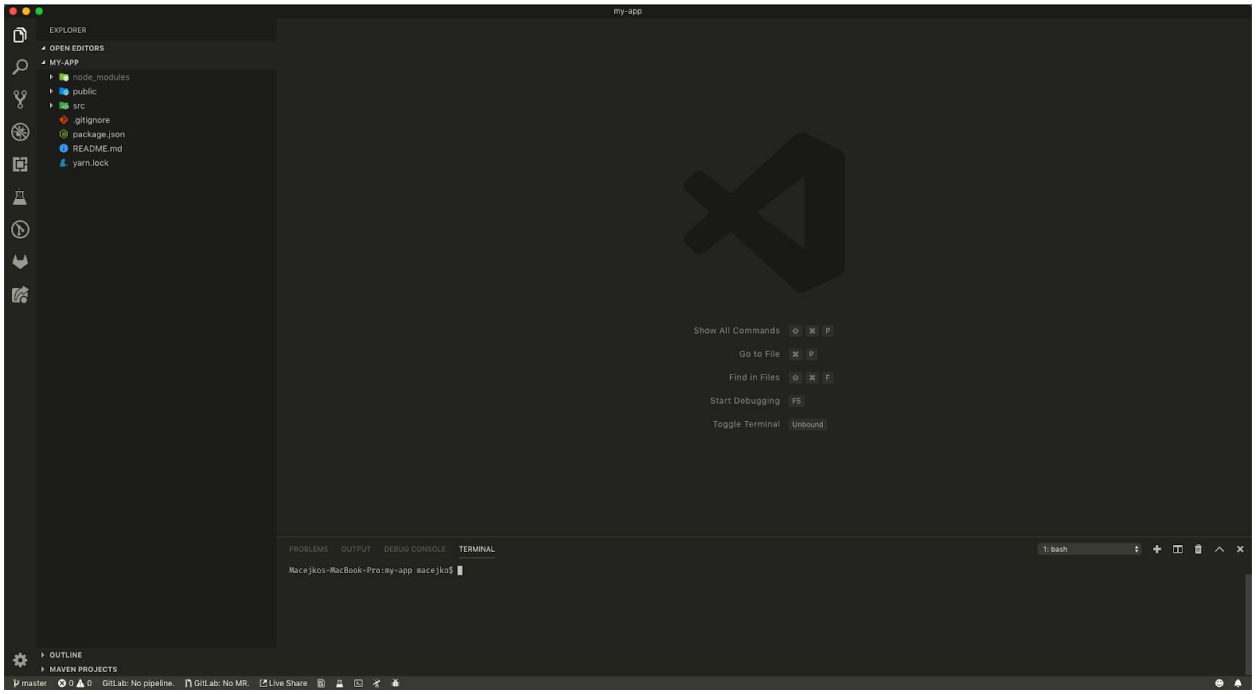
No configuration or complicated folder structure, you need only the files to build your application. Configuration is of course present, but it's hidden. You don't need to configure anything in your first steps, maybe in some advanced use cases.

In the project structure there are 2 main file entries, which have to exist with exact filenames.

- `public/index.html` - page template
- `src/index.js` - JS entry point

`package.json` is a file where there are stored all dependencies, scripts, versions and the description of your application. All the dependencies are installed to the folder `node_modules`, this folder should never be pushed to GIT (or the source control you use).
In `src` there are stored all the source codes for your application.

Now we can open the whole project in VS Code and everything should look similar to this screen.



Running your App

Now is the time to run your first application and see how it looks in the browser. Earlier it was mentioned that `package.json` also includes some scripts. These scripts are added by default, but you can also make your own.

First of all, there is a script to run your application (brace yourself, dunno if you can understand it):

```
npm start
```

There it is. In your browser (personally, I recommend Google Chrome) a new tab should open, with the url http://localhost:3000. This simple script starts running your local server, builds your source codes, and also watches the changes in the files. It automatically refreshes the window in your browser after a change is saved.

From now on you can edit the source code and the start of a heavy development can begin. But it's still only development, a production mode is a little different.
The important thing to understand is that in the file `/public/index.html`, that you can see on line 31, is the place where your application will be rendered.

```
<div id="root"></div>
```

This rendering is handled by the library `react-dom` and you can find it in the `index.js` on the line 7.

```
ReactDOM.render(<App />, document.getElementById('root'));
```

Build your App

To create a production build of your application, there is the following command.

```
npm build
```

After this command finishes, a new folder `build` is made in your structure. After uploading all this stuff to an http server, your application is accessible to the public. Note that all React source codes are compiled to JavaScript, into some files in `/build/static/js` and included in `/index.html`.

ES6 Modules

ES6 brings new options of how to use code between files. Before this official approach there were other solutions like RequireJs which allowed importing of code from other files and you can still sometimes see this approach in older code or code written for NodeJS environment.

```
// ES6 modules import
import App from './App';

// RequireJS import
const App = require('./App');
```

Default export

These serve to export the main variable of a given file (so for React components it usually is the component itself) and there can always be only one. In older import syntax such as `import * as App from './App';` they can be accessed through `.default` attribute.

```
const App = () => (
  <div className="App">
    {/* ... */}
  </div>
);
```

```
export default App;
```

Named export

They reveal your variables under specified names. In the CRA template, there are no named exports but we can add some as can be seen below.

```
export const a = 1;
export const b = 2;
export const c = 3;

const App = () => (
  <div className="App">
    {/* ... */}
  </div>
);

export default App;
```

Default import

You can import the App variable from the example above like this. It's also good to know that the name you give a default import does not need to match the name of the exported variable because the export is considered unnamed.

```
import App from './App';
import DefaultComponent from './App'; // will also work
```

Named import

To import named exports, wrap them in braces and separate with comma. Here the name must match the name of the exported variable. You can rename the variable locally with the keyword **as** followed by a new name.

```
import { a, b, c } from './App';
import { a as changedA } from './App'; // with a rename
import App, { a, b, c } from './App'; // with default import
```

Dynamic import

One final import option is to use the `import` function. This function is used to dynamically load modules during runtime. In React apps it can be used to f.e. load different components for

different page routes only when they are requested and React provides a useful `lazy` helper function together with the `Suspense` component for exactly that.

```
import { lazy, Suspense } from "react";
const LazyApp = lazy(() => import("./App"));

// Usage
<Suspense fallback={<div>Loading...</div>}>
    <LazyApp />
</Suspense>
```

JSX

JSX is a syntax extension to JavaScript. It is similar to HTML, but it allows you to integrate javascript code into it seamlessly. Most commonly, JSX is used in a return value of a function component (or render function in class component) but it can be used anywhere. Since JSX is an extension which browser's can't interpret, it has to be compiled into vanilla javascript. For this a JSX transform is used. Below is an example of simple component using JSX:

```
const Hello = () => (
  <div>Hello world!</div>
);
```

This code is then compiled into code below:

```
const Hello = () => /*#__PURE__*/React.createElement("div", null, "Hello
world!");
```

All the JSX code is transformed into React.createElement calls. This means that you don't really have to use JSX at all.

In a recent React release there was a small change to how JSX is compiled. Previously because it was compiled into a React.createEmelent call, you always had to have a **import React from 'react';** at the top of your file in order for it to work, but with the recent changes you no longer need to (full article).

Few examples of how JSX is similar but different to HTML:

```
// Javascript function as a prop
```

```
<button onClick={() => alert('Hello!!')}>Hello world!</button>

// className instead of class
<div className="box">Hello world!</div>

// Inline styles with camelCased names
<div style={{ display: 'flex', flexDirection: 'column' }}></div>

// Javascript variable as a child
<div>{attr}</div>
```

Styles and images in Create React App

Create React App provides different loaders for resource files such as css stylesheets or images that you will most likely need in your apps right out of the box.

CSS

You should already be familiar with the basics of CSS as it will not be part of this course. If you feel like your css knowledge needs some work you can start f.e. here.
To import css into the current file simply write an import statement like on below. This will make sure that the css will be bundled with the code and available during runtime. One such example can be seen in `App.js`.

```
import './App.css';
```

Images

You can also import static image files that you want to bundle into your application. They are different from files in the **/public** folder since these files will be copied during the build process into **/static/media** folder.

```
import logo from './logo.svg';

// Usage
<img src={logo} className="App-logo" alt="logo" />

// Resulting HTML
<img src="/static/media/logo.a7f48d81.svg" class="App-logo" alt="logo">
```

If you want to use images from the **/public** folder you can do so by using their relative url.

```
// Image from CRA template located in /public/logo512.png
<img src="/logo512.png" className="App-logo" alt="logo" />

// Resulting HTML
<img src="/logo512.png" class="App-logo" alt="logo">
```

Lastly there are SVG images that instead being used in an img node can be inlined into the DOM. This allows you to f.e. manipulate them with css.

```
import { ReactComponent as Logo } from './logo.svg';

// Usage
<Logo style={{ color: 'red' }} className="App-logo" alt="logo" />

// Resulting HTML
<svg
  viewBox="0 0 841.9 595.3"
  color="#61DAFB" // Original color from SVG
  class="App-logo"
  alt="logo"
  style="color: red;" // Our provided inline color
>
  {/* ... */}
</svg>
```

Note: In order for this example to take effect in the CRA template, you need to make a small change in logo.svg. Change the fill of g node to currentColor and add color attribute to the root svg node with the original value #61davb. This ensures that when applying inline styles the svg's color will be overwritten with what we provide.

Last thing generated by CRA and not explained yet is mysterious file `serviceWorker.js`. It's not necessary to dig deep, because it's disabled by default anyway, but it's a setup to help build your web application also available as offline-first app. It means that your app will be loaded faster, cached in browser.

UI Frameworks

UI Frameworks help you speed up your development by either providing out of the box fully featured components or tools and patterns that allow you to put together you own components easier. In the world of modern JavaScript are many (maybe too many?) different UI React libraries. Each of them has its own design style and ideology and it's up to you to find which one suits you the best. Here is a list of few frameworks:

- Material UI
- Styled system
- Libreact
- Semantic UI
- React Toolbox
- Ant Design
- React Bootstrap

CSS-in-JS

This pattern aims to bring css closer to how React (generally any JS UI library based on components which basically means all of them) code is managed and reused. Using cascading rules and selectors in the global context of a document is often tedious and doesn't bring anything useful to a web page built out of standalone reusable components. A component is not only the logic (js) and it's markup but also it's appearance. CSS-in-JS allows you to ditch css files, class naming conventions or fighting unexpected rule matches and replace it with a system similar to JSX compared to HTML, that allows you to seamlessly integrate javascript login into your styles.
There are multiple different implementations of this pattern that vary greatly in the execution. One of them is JSS which is also used by Material UI which we will be using during this course. It's similar to React's inline `style` attribute extended by many useful features. Other options are Emotion and Styled Components which both use tagged templates (feature of template string) to allow you to write css rules which can easily be modified by inline javascript.
There of course are many more CSS-in-JS libraries but these three we have tried personally and also have pretty active and big following.

Developer tools

In order to become efficient and understand what may be wrong in your code, it is essential you get to know the developer tools of your browser. We will be mainly using Chrome, so if you want to use another browser there may be some issues we won't have an answer to.

The developer tools are usually accessed by pressing F12 and contain multiple tabs of which for the start, most interesting to you will be **Elements** (Inspector in Firefox) and **Network** and later React's **Components** and **Profiler** tabs (from official extension).
Then there is the very helpful **inspect** button (*Ctrl* + *Shift* + *C*) that allows you to quickly find an element in a document (same for inspect button in React Components tab that picks React components).
By pressing **Esc** you can toggle the split-screen JS **Console** window (which will eventually contain your console.logs for debugging purposes). Here you can see the log of your application, access global variables or try out some smaller JS snippets.
You should definitely explore these tools on your own and get familiar with them because you will be looking at them a lot.